**International Journal of Information Security manuscript No.**
(will be inserted by the editor)

# On Improving Resistance to Denial of Service and Key Provisioning Scalability of the DTLS Handshake

**Marco Tiloca · Christian Gehrmann · Ludwig Seitz**

Received: date / Accepted: date

**Abstract** DTLS is a transport layer security protocol designed to provide secure communication over unreliable datagram protocols. Before starting to communicate, a DTLS client and server perform a specific handshake in order to establish a secure session and agree on a common security context. However, the DTLS handshake is affected by two relevant issues. First, the DTLS server is vulnerable to a specific Denial of Service (DoS) attack aimed at forcing the establishment of several half open sessions. This may exhaust memory and network resources on the server, so making it less responsive or even unavailable to legitimate clients. Second, although it is one of the most efficient key provisioning approaches adopted in DTLS, the pre-shared key provisioning mode does not scale well with the number of clients, it may result in scalability issues on the server side, and it complicates key re-provisioning in dynamic scenarios. This paper presents a single and efficient security architecture which addresses both issues, by substantially limiting the impact of DoS, and reducing the number of keys stored on the server side to one unit only. Our approach does not break the existing standard and does not require any additional message exchange between DTLS client and server. Our experimental results show that our approach requires a shorter amount of time to complete a handshake execution, and consistently reduces the time a DTLS server is exposed to a DoS instance. We also show that it considerably improves a DTLS server in terms of service availability and robustness against DoS attack.

**Keywords** Security · DTLS · Denial of Service · Key provisioning

## 1 Introduction

Secure communication has become particularly important for a great number of applications, ranging from e-commerce to plant monitoring, from certified e-mail to home automation. Currently, the *Transport Layer Security* (*TLS*) protocol [1] is the most widely deployed solution for securing network communication on top of reliable transport protocols, such as TCP [2]. However, since an increasing number of applications relies on datagram protocols, the IETF has recently introduced the *Datagram Transport Layer Security* (*DTLS*) protocol [3]. It is designed to be as similar to TLS as possible, but explicitly deals with the unreliable nature of datagram protocols, and thus can work also on top of UDP [4].

Like TLS, also DTLS requires two peers, namely *client* and *server*, to perform a *handshake* in order to establish a secure session. Typically, the handshake is started by the DTLS client, by sending a *ClientHello*

This work was carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme. The research leading to these results has received funding from the *European Union* Seventh Framework Programme (*FP7/2007-2013*) under *grant agreement* n° 246016.

M. Tiloca
SICS Swedish ICT AB, Security Lab
Isafjordsgatan 22, 16440, Kista (Sweden)
E-mail: marco@sics.se
Tel.: +46706046501 Fax: +4687517230
(Corresponding author)

C. Gehrmann
SICS Swedish ICT AB, Security Lab
Scheelevägen 17, 22370, Lund (Sweden)
E-mail: chrisg@sics.se

L. Seitz
SICS Swedish ICT AB, Security Lab
Scheelevägen 17, 22370, Lund (Sweden)
E-mail: ludwig@sics.se

message to the DTLS server. Then, the two peers are able to authenticate one another, agree on a common cryptographic suite, and establish the security material to be used for secure communication. Specifically, a key provisioning approach based on *pre-shared keys* (*PSKs*) [5] can be adopted, in order to avoid managing a Public Key Infrastructure (PKI) and to avoid the computational complexity introduced by public key cryptography operations. PSK has become very popular, and is particularly suitable to application scenarios such as building automation or smart metering, where the servers could potentially be even resource-constrained devices operating over low bandwidth networks.

In this paper, we argue that the DTLS handshake is affected by two relevant issues.

First, the DTLS server is highly vulnerable to a specific *Denial of Service* (*DoS*) attack. In particular, an adversary can repeatedly send *ClientHello* messages to the server, and force it to start performing a considerable number of handshakes. The only currently available solution against this attack relies on an optional and stateless *Cookie* exchange between client and server [3], performed in the initial phases of the handshake. However, this countermeasure only complicates the attack, and does not offer any fundamental protection against it. In fact, by intercepting handshake messages sent by the server, the adversary can induce the latter to establish a consistent amount of *half open* DTLS sessions. This can exhaust memory and network resources on the server, making it less responsive or even unavailable to process requests from legitimate clients. As an additional side effect, DoS attacks performed with valid spoofed addresses result in the server sending unexpected handshake messages to "innocent" nodes, with a consequent *amplification* effect against them. Therefore, the solution based on Cookie exchange is not a good countermeasure against a DoS performed by a well determined and resourceful adversary.

Second, in case the PSK provisioning approach is adopted, the DTLS server is supposed to store a set of cryptographic symmetric keys, each one of which is pre-shared with some associated clients. This approach is destined mostly to closed environments, where it is easily possible to provision shared keys to the involved parties. In a more dynamic environment, this means that a server would have to store and manage a considerable number of pre-shared keys, or, in a worst case scenario, even one for every possible client. Obviously, this does not scale well with the number of DTLS clients, and it considerably complicates key provisioning in dynamic application scenarios. However, the PSK scheme is very useful in a number of dynamic scenarios involving either constrained devices or users that do not have the capability to securely manage a PKI.

In this paper, we present a single and efficient security architecture which addresses both the two issues mentioned above, by smoothly and harmoniously combining the two following contributions. First, we define a possible alternative PSK scheme, namely *Derived Key Mode*, that prevents scalability and management issues on the server side, by drastically reducing the number of stored pre-shared keys to one only. This approach shifts the load of key management to a trusted third party and requires the client to do an extra round trip, thereby greatly reducing the load on the server. In addition, it makes the PSK scheme usable also in non closed, dynamic, environments, as a more lightweight alternative to approaches based on certificates and PKI.

Secondly, we describe our preventive solution to the DoS attack against the DTLS handshake. Our approach allows the server to identify invalid *ClientHello* messages, and promptly abort the handshake execution at the first step. So doing, the DoS attack is practically neutralized, by substantially limiting its impact against the server. Besides, any possible amplification effect against other nodes is prevented altogether, and the Cookie exchange is not required anymore, so avoiding one message round trip between client and server. The proposed security architecture relies on a *Trust Anchor* entity, which is assumed to be in a trusted relation with the DTLS server.

Furthermore, our approach displays the following benefits. First, it does not require changes to the DTLS standard and relies on a standardized extension method for *ClientHello* handshake messages. Second, it does not require any additional message exchange between DTLS client and server, so resulting in a communication overhead for the server which is lower than that when the Cookie exchange is adopted. Third, it does not significantly contribute to the computing overhead of DTLS client and server, i.e. the handshake process maintains the same order of computational complexity. Finally, in this paper we focus on the DTLS protocol, given its notably high vulnerability to DoS. Nevertheless, our proposal is deployable also in the TLS protocol without changing the actual standard, although the TLS handshake is much less exposed to DoS thanks to the preliminary TCP connection establishment.

In order to prove the validity of our approach, we did a proof of concept implementation, by extending the library *Scandium* [6], which implements DTLS 1.2 [3] in a stand-alone way. Then, we relied on our implementation to experimentally evaluate performance on the client and server side, considering the library *Californium* [7] and the *Constrained Application Pro-*

*tocol* (*CoAP*) [8] developed by the IETF working group *CoRE* [9]. We compare performance and effectiveness of our approach with those of the original DTLS handshake based on the Cookie exchange, considering also an actual DoS attack launched against a DTLS server. Results show that, in the presence of a DoS attack, our approach considerably improves a DTLS server in terms of robustness and service availability. Also, it consistently reduces the time a DTLS server is exposed to an attack instance, and requires a shorter amount of time to complete a handshake execution.

The rest of the paper is organized as follows. In Section 2, we review some related work on DoS attacks. Section 3 overviews the DTLS protocol, with particular reference to the handshake steps, while in Section 4 we highlight the handshake issues we address in the paper. Section 5 defines the application scenario we refer to, while we describe the provisioning of key material in Section 6, and discuss our *Derived Key Mode* scheme in Section 6.1. Then, Sections 7 and 8 detail our solution to DoS attack. In Section 9, we present our proof of concept implementation, discuss experimental results, and compare our approach with the original DTLS handshake based on the Cookie exchange. Finally, in Section 10 we draw our conclusive remarks.

## 2 Related Work

*Denial of Service* (*DoS*) is a well known attack aimed at making a host unavailable to its intended users, with the explicit intent to prevent them from accessing a service. It consists in exhausting some resource of the victim (e.g. network bandwidth), so preventing it from receiving legitimate service requests. With the help of more compromised hosts over the network or the Internet, such an attack can be mounted also in a coordinated and widely distributed fashion, i.e. *Distributed Denial of Service* (*DDoS*) [10], so resulting to be even more effective.

A considerable number of solutions to thwart DoS attacks have been proposed so far. As discussed in [11], they can be mainly classified into two categories, i.e. *router-based* and *host-based*. In particular, router-based solutions rely on defense mechanisms installed in IP routers in order to trace attack sources [12][13][14][15], or detect and block attack traffic [16][17][18][19][20][21]. The main drawback of router-based solutions is that they require not only router support, but also coordination among different routers and networks [11]. Besides, they typically rely on various IP traceback techniques based on *Probabilistic Packet Marking* (*PPM*) [13][22][23], which require to be universally deployed among *all* routers.

Conversely, host-based solutions locally work at victim hosts and are immediately deployable. The countermeasure against DoS proposed in this paper falls into this category. Most of current host-based approaches rely on resource management schemes [24][25], or aim at reducing resource consumption on the victim through different techniques, e.g. *Client Puzzles* [26][27], *SYN Cookies* [28] and DDoS-resilient scheduler [29].

*TCP SYN flooding* is one of the most common DoS attacks observed in the Internet [30]. Such an attack is not actually based on sending huge volumes of traffic to the designated victim, but is instead based on exploiting a weakness in the TCP connection establishment. In particular, spoofed TCP SYN packets are sent to the victim host, so triggering the execution of the TCP three-way handshake [2]. This induces the victim to transmit a TCP SYN-ACK packet and uselessly wait for the reception of the associated ACK packet. In such a way, the adversary can initiate, and leave unresolved, a large number of half open TCP connections on the victim, so exhausting its memory and network resources, and making it unable to serve other legitimate requests. This kind of attack is very similar to the DoS attack against the establishment of DTLS sessions that we address in this paper.

Different techniques to detect TCP SYN flooding attacks have been proposed. Most of them are based on identifying anomalies in TCP traffic, considering the arrival rate of bidirectional packets [31], asymmetries of traffic for both directions [32], or difference between the rates of TCP SYN packets and TCP FIN/RST packets [33]. However, as highlighted in [34], such approaches usually do not consider possible traffic variations, and manage to detect ongoing attacks only once the victim has been already seriously damaged. Then, [34] presents a mechanism for detecting SYN flooding traffic more accurately, by considering the arrival rate of SYN packets together with the time variation of arrival traffic.

Counteraction of TCP SYN flooding attacks has been investigated as well. In [35], Darmohray *et al.* discuss a router-based approach where routers mitigate the attack effects by sending SYN-ACK packets on behalf of the TCP server, and delivering SYN packets to the server only upon receiving the associated ACK packet from the TCP client. However, routers are required to handle the states of TCP connections on behalf of TCP servers, and thus become the actual victims of possible long term attacks. Conversely, host-based defenses typically rely on SYN Cache [36] and SYN Cookies [28][37]. The SYN Cache mechanism allows the victim to manage more half open TCP connections, by storing them in a global hash table rather than in a different backlog queue for each application.

Nevertheless, SYN Cache does not protect from SYN flooding attacks fundamentally. Instead, the approach based on SYN Cookies consists in encrypting the SYN packet header, and embedding the encryption output in the sequence number field of the SYN-ACK packet. Then, the TCP server allocates resources only upon receiving a valid ACK packet. However, as remarked in [34], the encryption process may become another weakness against the high-rated SYN packets. Also, the TCP server does not maintain any state of the TCP connection until the reception of a valid ACK packet, hence SYN-ACK packets cannot be retransmitted in case they are lost. Finally, SYN Cookies do not allow for encoding all TCP service parameters into SYN-ACK and ACK packets, so preventing clients from using TCP performance enhancements [27].

*Client puzzles* are another countermeasure against TCP SYN flooding [26][38]. Practically, they force TCP clients to solve a cryptographic riddle for each connection request, before the TCP server commits its resources. However, puzzles may result in a not negligible additional load on the client side, and it may be not easy to minimize such an impact by *tuning* their difficulty [27]. That is, there is the risk of introducing additional and annoying service delays for legitimate users.

In [39], Dean and Stubblefield consider a similar problem as we do, i.e. DoS attacks against the TLS handshake, aiming at exhausting server resources by inducing it to start and maintain half open TLS sessions. Their solution relies on client puzzles in order to make the attack more costly to be performed. In particular, the TLS server determines if it is overloaded with TLS connection requests by considering the amount of costly asymmetric cryptography operations performed lately. In such a case, the server asks clients to additionally solve puzzles during the handshake execution. However, [39] does not suggest other possible criteria to trigger the usage of puzzles, in case the TLS handshake does not rely on public key cryptography to establish security material between client and server. Also, it assumes the presence of an unconstrained server and unconstrained honest clients able to solve such puzzles. Hence, this can not be transferred to scenarios where potentially both client and server are resource constrained, whereas the adversary is not.

With particular reference to the DTLS handshake, the only currently available countermeasure against Denial of Service is based on a *Cookie* exchange performed during the first handshake phases [3]. This is reasonable and not surprising, since the current version of DTLS has become a standard protocol only in 2012 [3]. However, as we discuss in Section 4.1, the Cookie exchange does not protect from DoS attacks fundamentally, but only complicates their performance.

The solution we propose in this paper is an alternative to the Cookie exchange described in [3], and does not require any additional message exchange between DTLS client and server. Furthermore, it allows the victim server to quickly detect an ongoing DoS, in order to immediately abort invalid DTLS handshakes. Finally, it requires only the performance of lightweight computations on the client and server side. The experimental results we present in Section 9 show that our approach successfully counteracts a DoS attack launched against a DTLS server, so preserving service availability and proving to be more convenient and effective than the original approach based on Cookie exchange.

## 3 The DTLS protocol

This section briefly introduces the main aspects of the DTLS protocol considered in this paper. First, we provide an overview of the security services provided by DTLS. Then, we describe the DTLS handshake process, with particular focus on the message exchange between DTLS client and server. Finally, we discuss different available approaches to perform initial provisioning of security material to DTLS peers.

### 3.1 Overview

The *Datagram Transport Layer Security* (*DTLS*) protocol [3] has been designed by the IETF in order to provide secure communication for datagram protocols, such as UDP [4]. DTLS is based on the *TLS* protocol [1], and provides equivalent security guarantees, i.e. it allows client and server applications to communicate with one another preventing eavesdropping, tampering, and message forgery.

However, DTLS introduces some minimal changes with respect to TLS, in order to deal with the unreliable nature of datagram transport protocols. First, stream ciphers, such as *RC4* [40], cannot be adopted, and an explicit *sequence number* is included in every DTLS message. This makes distinct messages independent from one another, so allowing for correctly processing them despite the unreliable transport service and possible out-of-sequence delivery. Also, packet loss is explicitly addressed by means of local timeouts and message retransmission policies. Finally, upon receiving invalid messages, they can be silently discarded, and the associated DTLS connection may not be terminated.

Communication among two DTLS peers relies on secure *sessions*, identified by a unique *session ID* chosen

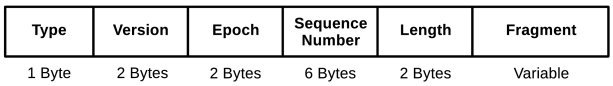


**Fig. 1** DTLS record format

by the DTLS server. Besides, messages are transmitted as a series of *records*, whose structure is shown in Figure 1. The *Type* field indicates the higher level protocol used to process the enclosed data, while the *Version* field states the employed version of the protocol. The *Length* field represents the size of the actual application data conveyed in the record, as a separate *Fragment* field. Finally, with respect to TLS, two additional fields are present, namely *Epoch* and *Sequence Number*. The former is incremented upon a possible change in the currently used security protocols and material. Instead, the latter is incremented for every new message transmitted by the same peer over the same DTLS connection. The concatenation of the *Epoch* and *Sequence Number* fields is considered as a single 64 bit fresh value, which is used to compute a Message Authentication Code for assuring integrity of protected DTLS records.

### 3.2 Handshake

A DTLS client and server establish a new secure session by performing a specific *handhshake* process. In addition, they can *resume* old previously established DTLS sessions, through a reduced handshake involving a shorter number of messages. The client is typically responsible for starting a session establishment, by sending a *ClientHello* message to the server.

Figure 2 depicts the message exchange occurring during a full DTLS handshake. In case multiple handshake messages are transmitted at the same step, they are grouped together in a single *Flight*. Messages whose name is reported among square brackets are optional or situation-dependent, and are not always sent. We refer the reader to [1][3] for further details about the DTLS handshake and the content of specific Flights.

As stressed in [3], the DTLS handshake is vulnerable to *Denial of Service* (*DoS*) attacks. That is, an adversary can repeatedly transmit *ClientHello* messages to a DTLS server, so triggering the establishment of new DTLS sessions. From the server perspective, this means allocating memory and network resources for new sessions' state, and performing useless processing operations. In order to address such an attack, DTLS introduces the optional exchange of a stateless *Cookie* value. That is, upon receiving the first *ClientHello* message, the DTLS server may reply with a *HelloVerifyRequest* message, including a locally generated Cookie.

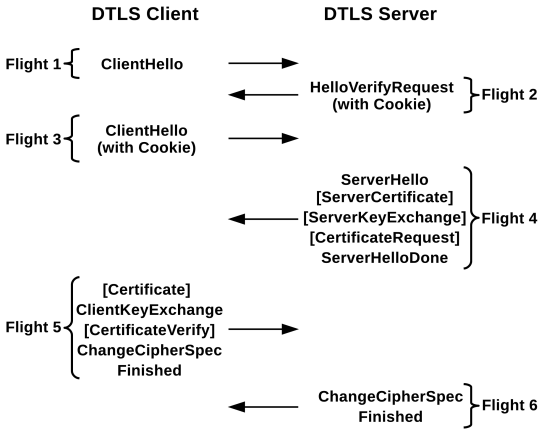


**Fig. 2** Full DTLS handshake message exchange

Upon its reception, the client must reply with a second *ClientHello* message, including the same Cookie received from the server. Then, the server proceeds with the rest of the handshake only if it successfully verifies the Cookie received in the second *ClientHello* message. This forces the adversary to receive the Cookie sent by the server, hence complicating attacks performed with spoofed IP addresses.

### 3.3 Key pre-provisioning

The DTLS handshake assumes that involved peers have been previously provided with some security material. This basically consists in a set of preinstalled keys used during the DTLS handshake to agree on a *premaster secret*. Such a *premaster secret* is used together with random values generated by the client and server to compute a *master secret*, from which the final security material is derived. In practice, DTLS admits two main approaches to provide preinstalled keys.

The first approach relies on *asymmetric key* pairs. In addition to the classical method based on *X.509* certificates [41], there also exist profiles for *raw public keys* [42], where key pairs come with no certificate, and may be generated by manufacturers and installed on nodes before deployment. In this case, a DTLS node relies on out-of-band means to validate raw public keys received from other peers, and typically retains a list of identities of peers it can communicate with.

The second approach relies on symmetric *pre-shared keys* [5]. In this case, a DTLS client shares a symmetric key with each DTLS server it may want to communicate with. During the DTLS handshake, a client indicates which particular symmetric key is going to be used, specifying a *PSK identity* in the *ClientKeyExchange* message. In order to help the client to select which identity must be used, the server can optionally

provide a *PSK identity hint* in the *ServerKeyExchange* message. Finally, both the client and server compute the *premaster secret* from the symmetric key they have agreed upon.

The latter approach is destined mostly to closed environments, where it is easily possible to provision shared keys to the involved parties, and has two main benefits. First, it makes it possible to avoid sending and receiving public certificates and performing costly public key operations, which is particularly important in the presence of resource constrained DTLS server. Second, it simplifies key management operations, especially in environments where connections are mostly configured manually in advance, and providing certificates is not considered a preferable or even feasible option. In the rest of this paper, we refer to the key provisioning based on pre-shared keys.

## 4 Weaknesses in the DTLS handshake

In this section, we discuss two issues of the DTLS handshake that we believe deserve to be addressed. That is, in Section 4.1, we describe a *Denial of Service* (*DoS*) attack based on the transmission of *ClientHello* messages, which can successfully be performed despite the *Cookie* exchange described in Section 3.2 is adopted. Then, in Section 4.2, we discuss the lack of scalability and resilience to dynamic scenarios of the PSK provisioning approach described in Section 3.3.

### 4.1 Denial of Service attack

As mentioned in Section 3.2, DTLS provides some protection against DoS attacks during the handshake execution, by introducing the exchange of a Cookie between client and server. However, such a mechanism is totally ineffective in case the attack is mounted with valid IP addresses [3], and only complicates the attack in case it is mounted with spoofed addresses. Thus, in the presence of a well determined and resourceful adversary, the DTLS handshake is practically still exposed to DoS attacks.

Hereafter, we consider an active adversary able to perform IP spoofing and to intercept messages sent by the DTLS server. While relying on address spoofing is not strictly required to perform a single attack instance, it makes it possible to: i) hide the location of the host(s) used to carry out the attack, so hiding a quick path to the adversary; and ii) perform the attack even when defenses based on address checking are adopted, e.g. lists of legitimate hosts or blacklists of untrusted hosts. On the other hand, the adversary must be somehow connected to the local network comprising the victim DTLS server, in order to intercept its replies to spoofed messages. In principle, the adversary must have access to the server's local network, and listen to network communication in promiscuous mode. This is particularly easy in case of physical proximity to insecure wireless networks. More generally, the adversary must have under her control at least one node in the same local network as the victim server. Then, the compromised entity can intercept messages sent by the server, and tunnel them to the adversary's host(s) actually responsible to perform the DoS attack.

As a possible alternative, the adversary may take advantage of the Internet Protocol's source routing option. This makes it possible to dictate the route that a reply message travels, e.g. through a network that the adversary can (more) easily control and where messages sent by the DTLS server can be conveniently sniffed. Although the source routing option can be disabled for security reasons, it is on the other hand a convenient choice for implementing mobility in IP networks.

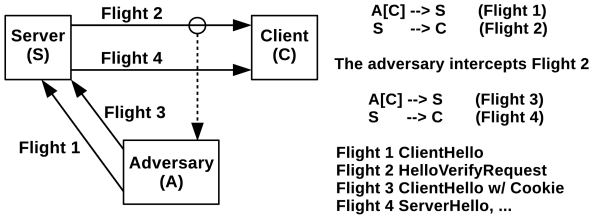


**Fig. 3** Denial of Service against DTLS server

In this paper, we consider a specific DoS attack aimed at inducing the server to continuously start new DTLS handshakes, in order to initiate and leave unresolved a large number of half-open DTLS sessions. More specifically, the adversary repeatedly sends *ClientHello* messages (Flight 1) to the victim DTLS server, so inducing it to start performing a handshake. Hereafter, we refer to such messages sent by the adversary as invalid *ClientHello* messages. Then, by intercepting Flight 2 and transmitting a Flight 3 including the expected *correct* Cookie, the adversary can induce the server to perform the DTLS handshake until Flight 4 has been transmitted, as shown in Figure 3. This means that, even upon preparing and sending Flight 4, the server is *not* able to assert whether the current handshake is genuine or not, i.e. whether a DoS attack is ongoing. Besides, the adversary can send multiple *ClientHello* messages at the same time, each one of which from a different spoofed address. This would force the server to handle multiple instances of invalid DTLS handshakes, so increasing the amount of performed processing opera-

tions, and possibly causing the exhaustion of memory and network resources. Hence, the server may become less responsive, or even unavailable, upon the reception of genuine *ClientHello* messages from legitimate clients. Finally, due to its amplification effect, such an attack can have a severe impact also on performance of "innocent" network nodes, which receive unexpected instances of Flight 2 and Flight 4 from the victim server.

Thus, we believe it is vital that the DTLS server is able to distinguish between valid and invalid *ClientHello* messages, so possibly aborting the DTLS handshake as soon as possible. In Section 7, we propose our preventive solution based on authenticated *ClientHello* messages, which allows the server to detect a DoS attack and halt invalid DTLS handshakes immediately after the reception of Flight 1.

As a final remark, in case the DTLS handshake relies on an asymmetric key pair, the adversary may also intercept Flight 4 and send a fake Flight 5 to the server. This would induce the server to process the fake *ClientKeyExchange* message and perform costly public key operations, so making the attack even more harmful. In the rest of the paper, we focus on the attack depicted in Figure 3, and consider the key provisioning based on pre-shared keys [5].

### 4.2 Drawbacks of pre-shared key provisioning

If the PSK provisioning scheme described in Section 3.3 is adopted, a DTLS server is required to store and manage a set of symmetric keys pre-shared with the respective DTLS clients. This may result in scalability issues on the server side, especially in a worst case scenario when each client is associated to a different key, or even represent a storage issue, in case the DTLS server is a resource constrained device with limited memory capabilities. Moreover, if the set of such clients dynamically varies over time, e.g. in a pay-per-use scenario, this would in turn require frequent re-provisioning of lists of trusted clients, and possibly pre-shared keys, to the individual servers. Hence, it is evident that the original PSK provisioning approach does not scale well with the number of clients, and may be a severe issue in terms of memory occupancy and complexity of key re-provisioning, especially in case of dynamic scenarios.

Other approaches have been proposed to address pre-distribution of shared key material. For instance, the authentication protocol *Kerberos* [43] relies on a trusted third party to establish shared keys among two parties having no previous security relation. Insofar, Kerberos has the same goal as our *Derived Key Mode* described in Section 6.1. However, Kerberos requires to perform a whole protocol consisting of three round trips, in order to authenticate both parties and establish a shared key. Also, Kerberos relies on *tickets* whose size is typically in the order of magnitude of 1 KB.

Another key management scheme, namely *Multimedia Internet KEYing* (*MIKEY*), has been first described in [44]. It is intended for real-time applications, and originally provided three provisioning modes, based on direct negotiation between peers, or pre-distribution of credentials, such as certificates. More recently, an additional ticket-based mode has been defined, namely *MIKEY-TICKET* [45]. It provides distribution of key material through a trusted *Key Management Service* (*KMS*), and is based on a ticket concept similar to that in Kerberos. Also, MIKEY-TICKET is particularly recommended for systems when an *initiator* peer may not know in advance the exact identity of the intended *responder* peer, or the set of possibly multiple responders changes over time. This is why, unlike in Kerberos, tickets are not bound to an exact identity until the actual responder becomes fully determined. However, this requires up to three different message round trips involving *three* different entitites. That is, the KMS is contacted also by responders, in order to *resolve* MIKEY tickets before providing the actual security material.

It is evident that the alternative pre-distribution approaches mentioned above do not have efficiency as their first goal, and are not primarily designed to work with DTLS. Instead, as discussed in Section 6.1, our *Derived Key Mode* scheme is integrated into the DTLS handshake, hence not requiring any extra message round trip, and uses a nonce value of roughly 40 bytes in size in order to establish a shared key between DTLS client and server. Also, as part of the security architecture presented in this paper, our scheme is effectively and harmoniously combined with the solution to DoS attack against DTLS we present in Section 7.

## 5 Application scenario

In the rest of the paper, we consider an application scenario where an adversary can easily perform the attack described in Section 4.1, so forcing a server node to uselessly start performing a DTLS handshake. Hence, we believe that a node acting as DTLS server should be able to *promptly* recognize invalid *ClientHello* messages, i.e. not sent by legitimate DTLS clients, and not further proceed with the DTLS handshake execution. Of course, at the same time, establishing a DTLS session must be possible to any legitimate DTLS client.

A possible way to address this consists in relying on a model where a DTLS client must obtain an authorization *before* contacting a DTLS server to start the DTLS

handshake and establish a secure connection. In the following, we assume that such an authorization process is entrusted to a dedicated *Trust Anchor* (*TA*) entity. In particular, we assume that the implemented policies allow the *TA* to effectively determine whether to issue an authorization to a requesting, legitimate, client. Furthermore, the *TA* can generally provide additional services. For instance, it can also act as an *Authorization Service* managing permission release to access different resources with different access rights [46][47], or as a *Key Distribution Center* providing security material, so avoiding the introduction of a dedicated key management infrastructure. With respect to an approach based on proxy servers, this model has the advantage to not require any particular adaptations to the actual communication between clients and servers.

The *TA* can be practically implemented as an actual centralized entity, or according to a distributed architecture. The centralized approach is easier to be implemented and likely to be more efficient. At the same time, a purely centralized *TA* can constitute a single point of failure and be an easier target for a number of security attacks. Yet, it is reasonable to assume that the *TA* is a special-purpose computer properly designed, implemented and managed to be reliable and secure. Although server reliability and security are still research issues, the literature provides a number of established techniques and methodologies, e.g. [48][49][50]. Note that it is reasonable to rely on such techniques to protect relatively few deployed *TA*s. Instead, it is impractical to adopt them on a large scale for any host possibly acting as DTLS server, or even unfeasible in case of resource-constrained servers.

On the other hand, adopting a distributed architecture is beneficial in terms of robustness and availability, and avoids a single *TA* instance from being a single point of failure. This requires to synchronize sequence number values and long term keys $K_{MS}$ between the different *TA* replicas. This work is not devoted to any specific approach for synchronizing *TA* replicas, whose choice should take into account the architectural, application-level, and security constraints of the very infrastructure and domain the *TA* belongs to. Further details about the actual authorization process performed on the *TA* and practical architectural design choices are out of the scope of this paper.

In the following, we refer to the application scenario in Figure 4, and consider the presence of three distinct entities, namely a DTLS server *S*, a DTLS client *C*, and the Trust Anchor *TA*. In particular, we assume that the *TA* is trustworthy and thus cannot be compromised by an attacker. In addition, we consider *S* as associated to this *TA* only, according to a mutual trust relation. Also,

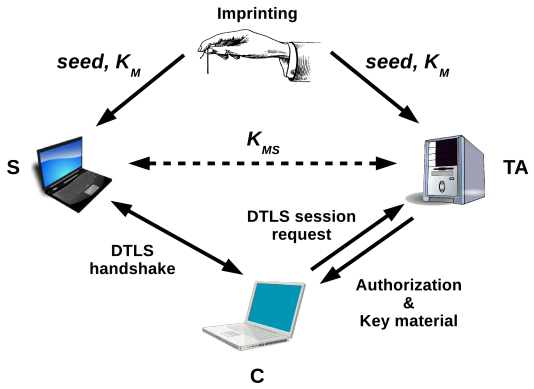


**Fig. 4** Application scenario

we assume that client *C* can rely on a service such as the IETF Resource Directory [51] to know what is the specific Trust Anchor *TA* associated to server *S*.

Finally, the communication between the *TA* and *C* is required to be properly secured. To this end, a possible straightforward approach relies on establishing a TLS/DTLS session between the *TA* and *C*. On one hand, this would move exactly the same DoS issue discussed in Section 4.1 from the server *S* to the *TA*. However, while it would be clearly impractical for every generic host configured as DTLS server, the relatively few deployed *TA*s can be implemented in such a way to be adequately reliable and robust against the considered DoS attack, as previously discussed in this section. On the other hand, the same key administration issues discussed in Section 4.2 would be moved from the server *S* to the *TA*. However, unlike generic, possibly storage-constrained, hosts configured as DTLS servers, it is reasonable that a *TA* is a special-purpose entity, provided with plentiful of resources, and thus able to properly manage a non negligible amount of pre-shared keys. Besides, offloading DTLS servers from having to manage a large number of shared secret keys is beneficial from a whole system perspective, as it scales better than provisioning pairwise shared keys between any possible pair of clients and servers.

Nevertheless, in Section 6 we refer to a possible alternative approach, in order to secure the communication between *C* and the *TA* without relying on any pre-established association between the two parties.

## 6 Provisioning of security material

In this section, we describe an approach to generate the necessary security material between *C* and *S*. Our approach has the following benefits. First, it limits the amount of generated security material, so minimizing the number of involved cryptographic keys. Second, the

security material is required to be *provided* to client $C$ only, while server $S$ can implicitly *derive* it, thus minimizing transmissions.

Hereafter, we assume that all entities $C$, $S$, and the $TA$ agree on a pseudo-random function $PRF(\cdot)$, which produces an output whose size is 256 bits. In particular, the considered $PRF(\cdot)$ function is based on a *HMAC* function [52], and relies on the same data expansion scheme adopted by DTLS and described in [1]. Furthermore, $S$ and the $TA$ secretly share i) a value *seed*; ii) a long term symmetric *master key* $K_M$ which is only used to generate other security material; and, finally, iii) a symmetric *master session key* $K_{MS}$ which is computed as $K_{MS} = PRF(K_M, seed)$. Specifically, we assume that *seed*, $K_M$, and $K_{MS}$ are 256 bits in size, and that *seed* and $K_M$ have been pre-established during the initial configuration phase of $S$, also known as *imprinting* [53]. An exhaustive list of the security material referred throughout the paper is reported in Appendix A.

Before starting to perform a DTLS handshake with $S$, client $C$ must first contact the $TA$, in order to be authorized to proceed further. In case the $TA$ accepts such a request, it provides $C$ with four pieces of information, as described below. This section considers a possible approach to secure the communication between $C$ and the $TA$. However, the adoption of alternative procedures is, of course, possible and left open.

We denote a message $M$ sent by host $A$ to host $B$ and conveying a quantity $Y$ as "$M: \ A \rightarrow B \quad Y$". Also, by $\{x\}_K$ we denote the encryption of a quantity $x$ by means of key $K$. We assume that cryptographic primitives are secure, and secrets have a size that discourages an exhaustive search. Finally, we denote by $N_C$ and by $K_{C-TA}$ a fresh nonce and a symmetric key randomly generated by client $C$, respectively.

Upon contacting the $TA$ for the very first time, $C$ establishes a long term secret key $K_{C-TA}$ with the $TA$. For instance, $K_{C-TA}$ can be established through the procedure described in Appendix B, considering the $TA$'s public key. Note that such a key establishment will not be necessary when $C$ contacts again the same $TA$, regardless the specific DTLS server involved in the handshake to be performed thereafter.

After having established $K_{C-TA}$, and before opening a new DTLS session with a server $S$, client $C$ performs the following message exchange with the $TA$.

$$M1: \ C \rightarrow TA \quad < C, S, N_C >$$
$$M2: \ TA \rightarrow C \quad < \{SN, N, K_S, K_{S-C}, N_C\}_{K_{C-TA}} >$$

Upon receiving message $M1$, the $TA$ sends message $M2$ to client $C$, encrypting it by means of key $K_{C-TA}$, and including the nonce $N_C$ retrieved from message $M1$. Upon receiving message $M2$, client $C$ can retrieve its content by means of key $K_{C-TA}$, and verify that the message is fresh thanks to the presence of nonce $N_C$. Then, $C$ retrieves four pieces of information, namely $SN$, $N$, $K_S$ and $K_{S-C}$.

In particular, $SN$ is a 32 bit sequence number assigned to $S$ and managed by the $TA$. A given sequence number value is associated to a specific DTLS session, and does not change over time. The nonce $N$ is a string composed of a fixed tag containing information about the issuer, recipient and target for this nonce, and the sequence number $SN$. Finally, two 256 bit symmetric keys are provided, namely $K_{S-C} = PRF(K_{MS}, N)$ and $K_S = PRF(K_{MS}, SN)$. The key $K_{S-C}$ is used during the DTLS handshake between client $C$ and server $S$ (see Section 6.1), while $K_S$ is used to protect server $S$ from DoS against the DTLS handshake (see Section 7). If this exchange succeeds, the $TA$ increments the sequence number associated to the relevant server $S$.

Note that, in a practical implementation, it is important to guarantee that the $TA$ does not exceed a certain maximum rate when issuing sequence number values, in order to prevent an attacker with legitimate credentials from quickly consuming the sequence number space associated to the relevant $S$, and thus making the $TA$ unable to correctly serve other clients.

As discussed in Section 5, a distributed implementation of the $TA$ would require synchronization between the different $TA$ replicas. In particular, synchronization procedures may take time to propagate changes to all the replicas. Yet, we believe that synchronizing a 4-byte integer value between the different $TA$ replicas is not likely to result in a considerable impact in terms of network latency. That is, synchronizing the sequence number value among all the $TA$ replicas is likely to be affordable even after every single sequence number value has been issued, before proceeding with issuing the next value.

### 6.1 Derived Key Mode

In the following, we present our key provisioning scheme based on the PSK approach [5], namely *Derived Key Mode*. Basically, client $C$ considers $K_{S-C}$ as the *pre-shared key* (to be) shared with server $S$, and uses it to generate the *premaster secret*. Then, while performing the DTLS handshake and preparing Flight 5, client $C$ writes the nonce $N$ in the *PSK identity* field of the DTLS *ClientKeyExchange* message (see Figure 2).

Upon receiving the DTLS *ClientKeyExchange* message, server $S$ retrieves the conveyed nonce $N$ from the *PSK identity* field. After that, $S$ does not retrieve a pre-shared symmetric key associated to client $C$, as usually

assumed by the PSK approach. Instead, $S$ uses nonce $N$ to compute the key $K_{S-C}$ as $K_{S-C} = PRF(K_{MS}, N)$. Then, the server considers $K_{S-C}$ to be the *pre-shared key* shared with client $C$, and uses it to generate the *premaster secret*. The latter is then used to derive the *master secret* in order to generate the actual DTLS security material (see Section 3.3).

The *Derived Key Mode* described above has the following benefits. First, it makes the PSK scheme usable also in non-closed, dynamic, environments, where potential DTLS clients and servers do not necessarily have an established security context. Second, it does not require to provide server $S$ with multiple *pre-shared keys* through any out-of-band provisioning methods. Third, it avoids key re-provisioning of server $S$ in case the set of potential DTLS clients changes over time. This makes it possible to manage dynamic trust relations without re-provisioning individual DTLS servers. Fourth, since the considered $PRF(\cdot)$ function relies on the same data expansion scheme adopted by the DTLS handshake, computing the key $K_{S-C}$ does not significantly impact on performance, i.e. the overall handshake maintains the same order of computational complexity. Finally, our approach requires a server to store only the key $K_{MS}$ shared with the *TA*, so avoiding key management issues and scaling well with the number of DTLS clients.

These benefits are achieved at the cost of moving some load from server $S$ to the *TA* and client $C$, which needs to perform additional communication with the *TA*. We believe that this is a good trade-off, which expands the applicability of the PSK scheme beyond pre-provisioned shared keys, so making it possible to use it in scenarios with a large number of dynamically changing communication partners, as could be found in building automation or smart metering use cases.

The *Derived Key Mode* also requires the provisioning of key material between clients and the *TA*, as well as between servers and the *TA*. This approach obviously scales better than provisioning pairwise shared keys between all clients and servers. Furthermore, the key material shared between servers and *TA*s can be provisioned statically upon the enrollment of servers, without any assumptions on which client will need to access which server.

## 7 Counteracting DoS attack

In this section, we present our approach to protect a DTLS server from the DoS attack discussed in Section 4.1. Our proposal represents an alternative to the standard Cookie approach described in [3], and is based on sending a *single* and authenticated *ClientHello* message upon initiating the DTLS handshake. Specifically, our goal is to allow a DTLS server to detect invalid *ClientHello* messages, and abort the associated DTLS handshakes as soon as possible. To this end, we rely on the key $K_S$ provided to client $C$ by the *TA* and derived by server $S$ upon the reception of a *ClientHello* message.

Note that our countermeasure to DoS entirely takes place during the first step of the DTLS handshake, and is thus agnostic of following message exchanges. Hence, it can be adopted in the presence of any key provisioning method considered by the DTLS client and server later during the handshake. In the rest of this section, we focus on the establishment of a *new* DTLS session. For the reader's convenience, we discuss minor differences during DTLS session resumption in Appendix D.

Upon starting a DTLS handshake, client $C$ includes a *lightweight* and *short* Message Authentication Code (MAC) in the outgoing *ClientHello* message. Then, by checking the validity of the conveyed MAC, server $S$ is able to promptly assert whether the received *ClientHello* message is *genuine* or not, i.e. if it has been sent by a legitimate DTLS client. In such a case, the handshake can regularly proceed, otherwise a DoS attack is assumed to be currently ongoing and the *ClientHello* message is silently discarded. Unlike the Cookie exchange, such a procedure does not require any additional handshake messages, so limiting the communication overhead on both the client and server side. Also, it avoids possible amplification effects against other network nodes altogether.

The MAC mentioned above can be computed using different types of standard algorithms and principles, such as common HMAC functions [52] considered by the DTLS protocol itself. However, most standard MAC algorithms display a relatively long computation time and produce output which is non negligible in size, hence introducing a significant communication overhead. Performing a simple truncation of computed output is not a recommended solution, since it would surely limit such an overhead, but would also reduce MAC security, especially in terms of robustness against forgery. In order to overcome such issues, our approach refers to an unconditionally secure MAC construction, which relies on universal hashing based on a *Galois Field* multiplication construction [54][55]. This construction has the specific advantage to assure a sufficient low forgery probability also in case of MACs which are small in size. In particular, hereafter we refer to a MAC construction whose output is only 16 bits in size. More details about the actual MAC computation process are provided in Section 7.2.

In the rest of this section, we first discuss the procedure used to authenticate *ClientHello* messages, and then describe the actual MAC computation process.

### 7.1 ClientHello message authentication

In the following, we define a *SecureHandshake* Hello Extension for *ClientHello* messages, according to the guidelines provided in [1]. To this end, we define an *Extension* structure, as reported below.

```
struct {
  ExtensionType extensionType;
  ExtensionSize extensionSize;
  ExtensionData extensionData <0..2^16-1>;
} Extension

struct {
  uint32 sequenceNumber;
  uint16 resumptionCounter;
  uint16 helloMAC;
} ExtensionData
```

We also introduce the *ExtensionData* structure as value for the *extensionData* field. This structure includes a *sequenceNumber* field, containing the value *SN* which was provided to *C* by the *TA* (see Section 6). We believe that a 32-bit field results in an acceptable and fairly long amount of time, before the sequence number space associated to a DTLS server gets exhausted, and a new long term key $K_{MS}$ has to be established between the DTLS server and the *TA* (see Appendix C). The *resumptionCounter* field is used to provide replay protection, in case of DTLS session resumption (see Appendix D). That is, it univocally identifies the next resumption instance associated to a given DTLS session already established between *C* and *S*. We believe that a 16-bit field adequately accommodates most of the DTLS clients' needs to resume a previously opened session. Finally, the *helloMAC* field contains the computed MAC associated to the *ClientHello* message. We believe that a 16-bit MAC has a reasonable size to discourage the DoS attack considered in Section 4.1. Also, we would like to point out that the purpose of this MAC is not to provide any authenticity of the whole *ClientHello* message, but only to be a deterrent against the considered DoS attack. If we assume that both the *extensionType* and *extensionSize* fields are 2 bytes each in size, then our Hello Extension is overall 12 bytes in size.

Before starting a DTLS handshake with server *S*, client *C* performs the following steps.

1. Client *C* contacts the *TA*, and receives a sequence number *SN* and a session key $K_S$ (see Section 6).
2. Client *C* derives a key $K_{MAC}$ which will be used to compute the MAC for the *ClientHello* message. In particular, $K_{MAC} = PRF(K_S, "new\_session")$.
3. Then, client *C* creates an instance of the *SecureHandshake* extension presented above, and includes it in the *ClientHello* message to be sent to *S*. The *sequenceNumber* and *resumptionCounter* fields are initialized to *SN* and 0, respectively.
4. Client *C* computes a MAC *v*, relying on the key $K_{MAC}$ and a Galois Field multiplication construction based on a 16 bit Galois Field [54]. The MAC computation takes as input also our *SecureHandshake* extension, although only the *sequenceNumber* and *resumptionCounter* fields are considered.
5. Finally, client *C* writes the computed MAC *v* in the *helloMAC* field of the *SecureHandshake* extension.

After that, client *C* starts the DTLS handshake by sending the *ClientHello* message to server *S*. Client *C* is supposed to store key $K_S$ in case the associated DTLS session might be resumed in the future. The key $K_{MAC}$ is discarded, as a new different one will be generated in case of session resumption (see Appendix D).

Upon receiving the *ClientHello* message, server *S* retrieves the *SecureHandshake* extension, and performs the following steps.

1. First, server *S* checks that the *resumptionCounter* field is set to 0, in order to verify that the received message is consistent with the establishment of a new DTLS session.
2. Then, server *S* retrieves *SN* from the *sequenceNumber* field, and computes $K_S = PRF(K_{MS}, SN)$ and $K_{MAC} = PRF(K_S, "new\_session")$. Since the considered $PRF(\cdot)$ function relies on the same data expansion scheme adopted by the DTLS handshake itself and described in [1], computing the keys $K_S$ and $K_{MAC}$ does not significantly impact on performance, i.e. the overall handshake maintains the same order of computational complexity.
3. Then, server *S* computes a MAC $v^*$ by means of $K_{MAC}$ and the Galois Field multiplication construction, taking as input the whole *ClientHello* message but the *helloMAC* field of the *SecureHandshake* extension.
4. Finally, server *S* compares the resulting MAC $v^*$ with the MAC *v* carried within the *SecureHandshake* extension. In case of negative match, the message is considered invalid and is silently discarded. Instead, in case of valid MAC, server *S* assumes that the message has been sent by a legitimate client, and continues to perform the DTLS handshake *without* any Cookie exchange with client *C*, i.e. *S* proceeds with the transmission of the *ServerHello* message.

### 7.2 MAC computation

In this section, we describe the actual computation of the 16 bit MAC used to authenticate *ClientHello* mes-

sages (see Section 7.1). Note that the procedure discussed in this paper is only one among several possible ways to perform the MAC computation. Nevertheless, since the resulting output is an unconditionally secure MAC [56], and the algorithms used to produce it have been extensively studied in terms of correctness and complexity [54][55], we strongly believe that the suggested method is close to be optimal in terms of simplicity and computational efficiency. Therefore, it does not significantly contribute to the computing overhead of DTLS client and server, i.e. the handshake process maintains the same order of computational complexity.

Having defined the MAC to be 16 bits in size, we assume that elements involved in the MAC computation are 16 bits in size as well. This is done only for the sake of simplicity in the following description, while it is clearly possible to rely on elements of different sizes. Also, we denote $GF(2^{16})$ as a Galois field with a size of 16 bits [54], and define $a, b, c \in GF(2^{16})$ as 16 bit key values, which are valid only for the establishment (or resumption) of a single specific DTLS session. Given the key $K_{MAC} = PRF(K_S, "new\_session")$ introduced in Section 7, the keys $a$, $b$, and $c$ are defined as follows. Key $a$ coincides with the bits of $K_{MAC}$ ranging from position 0 to position 15; key $b$ coincides with the bits of $K_{MAC}$ ranging from position 16 to position 31; key $c$ coincides with the bits of $K_{MAC}$ ranging from position 31 to position 47.

Now, let us consider the *ClientHello* message $M$ as divided into a number of equally sized chunks of 16 bits each, and refer to the $i$-th chunk as $m_i$. Then, assuming message $M$ to be $b_M$ bits in size, we can represent it as the concatenation of $n = \lceil (b_M/16) \rceil$ chunks, i.e. $M = \{m_0||m_1||...||m_{n-1}\}$, where $m_0, m_1, ..., m_{n-1} \in GF(2^{16})$. Hence, the 16 bit MAC $v$ associated to message $M$ is computed as

$$v = (m_0 + a \cdot m_1 + ... + a^{n-1} \cdot m_{n-1}) \cdot b + c$$

## 8 Replay protection

Let us assume that an adversary intercepts and stores a valid *ClientHello* message, i.e. including the *SecureHandshake* extension, which has been previously sent to $S$ by a legitimate client. Then, she can retransmit such an old message to $S$, which would consider it valid and proceed to perform the DTLS handshake up to Flight 4. In this section, we discuss a possible way to address such an issue, and protect our countermeasure from replay of old *ClientHello* messages. Specifically, in the following we refer only to the establishment of *new* DTLS sessions. For the reader's convenience, Appendix D describes how our approach can provide replay protection during resumption of old DTLS sessions.

We assume that server $S$ relies on a sliding window mechanism defined as follows. Let us denote a sliding window $W$ of size $A$ as a pair $\{w, w_b\}$. Specifically, $w$ is a vector composed of $A$ bits, thus requiring $\lceil (A/8) \rceil$ bytes. Instead, $w_b$ indicates the current left bound of the window $W$. That is, upon receiving a *ClientHello* message, $w_b$ represents the lowest acceptable value carried in the *sequenceNumber* field of the *SecureHandshake* extension. Upon $S$ startup, $w_b$ as well as all bits of $w$ are initialized to 0.

Note that the $A$ value should be chosen according to the expected frequency of DTLS session requests on server $S$, and in order to deal as best as possible with the unreliable message delivery due to datagram transport protocols. Of course, the larger the sliding window, the more accurate and resilient is the protection against replay attacks, but the greater the amount of required memory on the server side.

Upon receiving a *ClientHello* message aimed at establishing a new DTLS session, i.e. the *session ID* field is empty, server $S$ retrieves the sequence number $SN$ from the *sequenceNumber* field of the *SecureHandshake* extension. Then, the following checks are performed.

**Case 1**. If $SN < w_b$, the message is considered too old and is silently discarded.

**Case 2**. If $w_b \leq SN < min(w_b + A, 2^{32})$, $S$ defines $i = (SN - w_b)$, and checks the $i$-th bit of vector $w$. If such a bit is set to 1, i.e. the same $SN$ has been previously used, then the received message is considered to be a replay and is silently discarded. Instead, if such a bit is set to 0, $S$ proceeds with the *ClientHello* message processing including the MAC verification, as described in Section 7.1. Then, in case the message is invalid, it is silently discarded. Otherwise, $S$ continues to regularly perform the DTLS handshake.

**Case 3**. If $(w_b + A) \leq SN < 2^{32}$, $S$ proceeds with the *ClientHello* message processing including the MAC verification, as described in Section 7.1. In case the message is invalid, it is silently discarded. Otherwise, $S$ continues to regularly perform the DTLS handshake.

Once the DTLS handshake has been completed, $S$ checks whether the condition $SN \geq w_b$ is still valid. In such a case, the sliding window $W$ is updated as follows.

**Case A**. If $w_b \leq SN < min(w_b + A, 2^{32})$, $S$ defines $i = (SN - w_b)$ and sets the $i$-th bit of vector $w$ to 1, so

marking the sequence number $SN$ as used.

**Case B**. If $(w_b + A) \leq SN < 2^{32}$, $S$ defines $w_{new}$ as $w_{new} = (SN - A + 1)$. Then, $S$ updates vector $w$ as $w = w \gg (w_{new} - w_b)$. Specifically, "$\gg$" is the unsigned right bit shift operator, i.e. the leftmost position of vector $w$ is filled with 0. After that, $S$ updates $w_b$ as $w_b = w_{new}$. Finally, $S$ defines $i = (SN - w_b)$ and sets the $i$-th bit of vector $w$ to 1, so marking the sequence number $SN$ as used.

Once all possible values of $SN$ have been used on the *TA*, the latter has to provide server $S$ with a new key $K^+_{MS}$ (see Appendix C). However, it is possible that when it receives such a new key $K^+_{MS}$, the server $S$ has not yet received all *ClientHello* messages associated to the $SN$ remaining values. In this case, $S$ saves the current window $W$ and key $K_{MS}$, as $W^* \leftarrow W$ and $K^*_{MS} \leftarrow K_{MS}$. Then, the sliding window $W$ is reinitialized, i.e. all bits of vector $w$ are set to 0 and $w_b = 0$. After that, new *ClientHello* messages with fresh $SN$ values $\{0, 1, ...\}$ will be processed, referring to the reinitialized sliding window $W$, and the new key $K^+_{MS}$. However, for a given amount of time $T$, the server $S$ considers also late *ClientHello* messages conveying an $SN$ value $x$ such that $w^*_b \leq x < 2^{32}$. Such messages are processed by referring to the sliding window $W^*$ and the key $K^*_{MS}$, according to the same update procedure described above. Of course, $T$ is supposed to be much less than the amount of time practically needed to observe two consecutive wrap arounds of $SN$ values on the *TA*.

## 9 Experimental evaluation

In order to evaluate performance of our approach, we did a proof of concept Java implementation of the additional security services described in Sections 6.1, 7, and 8. Specifically, we referred to the *Constrained Application Protocol* (*CoAP*) [8], a lightweight application protocol designed by the IETF working group *CoRE* [9], which explicitly relies on DTLS to provide secure communication, if requested. In particular, we considered the Java library *Californium* [7], which provides a full implementation of the *CoAP* protocol. Also, we properly extended the Java library *Scandium* [6], which implements DTLS 1.2 [3] in a stand-alone way, although it has been primarily designed to work together with *Californium* on top.

In the following, we discuss the resulting memory footprint of our implementation, as well as the overall transaction length experienced on the client side and the processing overhead introduced by our approach. Also, we evaluate the effectiveness of our approach in the presence of an actual DoS attack launched against a DTLS server.

Compared with the original version of *Californium* and *Scandium*, our security services, even without any optimizations, result in additional 23.17 KB (+5.99%) and 21.45 KB (+5.57%) of memory on the client and server side, respectively. This suggests that an optimized implementation in C or Assembly-like languages is very likely suitable to memory constrained platforms.

In order to evaluate our approach, we ran a set of experimental tests on an ethernet local network, considering either a system relying on the original DTLS protocol based on the Cookie exchange, or a system relying on DTLS together with our additional security services. More in detail, first we ran our Java based test environment relying on the *Cookie* exchange mechanism and the original pre-shared key establishment based on the PSK approach [5] described in Section 3.3. Then, we relied on the same setup and performed the very same experiments, but in the presence of our security services, i.e. the *Derived Key Mode* scheme presented in Section 6.1, and the protection against the DoS attack described in Sections 7 and 8. Such sets of experiments allow us to compare our security services with respect to a system based on the original DTLS protocol, in terms of transaction length, processing overhead, and effectiveness against DoS attacks. All results have been averaged over 20 independent repetitions, and confidence intervals have been derived, with 95% confidence level.

In the presence of our security services, we also consider the preliminary interaction between $C$ and the *TA* to provide the client with all the necessary security material (see Section 6). Then, we consider the following simple application. First, the DTLS client $C$ and server $S$ establish a secure connection by performing a DTLS handshake. Then, client $C$ sends a *CoAP GET* request to server $S$, which replies with a *CoAP* response message whose payload size is set to 623 bytes. Finally, client $C$ terminates the DTLS session with server $S$.

Furthermore, we relied on the DTLS cryptosuite TLS_PSK_WITH_AES_128_CCM_8 [57], which assumes the execution of a pre-shared key DTLS handshake, is based on a single *authenticated encryption* operation, and provides both confidentiality and data origin authentication. Also, the adopted pseudo-random function $PRF(\cdot)$ is based on the specific hash function *SHA-256* [58]. As to the preliminary interaction between client $C$ and the *TA*, we referred to the message exchange described in Section 6, and considered *RSA* [59] and *AES* [60] when protecting messages $M1$ and $M2$, respectively.

Besides, the server $S$ implements a handshake timeout, i.e. a session establishment is aborted in case the

DTLS hansdhake is not successfully completed within a maximum amount of time. This forces the adversary to continue performing the attack even after the maximum amount of (half) open sessions has been reached, in order to keep the server unavailable.

To perform our tests, we considered generic hardware platforms running Java SE runtime environment (version 1.8.0_45). More in detail, the host acting as DTLS client $C$ was a common desktop PC with 4 GB of RAM and an Intel i5-3570 CPU, while the DTLS server $S$ was a laptop PC with 4 GB of RAM and an Intel i5-3317U CPU. The host acting as the *TA* was a desktop PC with 4 GB of RAM and an Intel Core2 CPU. Finally, in case of attack performance, the host acting as the adversary was a desktop PC with 8 GB of RAM and an Intel i7-3517UE CPU.

In the following, we first consider an attack-free scenario, and separately present evaluation results referred to the client and server side, in Sections 9.1 and 9.2, respectively. Then, we provide a discussion of such results in Section 9.3. Finally, in Section 9.4, we consider the execution of an actual DoS attack against server $S$, in the presence of either the original DTLS protocol or our security services, and discuss attack effects on the server robustness and availability.

### 9.1 Client results

In this section, we present our experimental results referred to client $C$. In particular, the following metrics have been considered on the client side.

**ClientHello preparation**. Time spent by $C$ to prepare the *ClientHello* message. In case the original DTLS protocol is considered, this time refers to the first *ClientHello* message generated and sent by $C$. Instead, in the presence of our security services, this time encompasses also the preparation of the *SecureHandshake* extension.

**ClientKeyExchange preparation**. Time spent by $C$ to prepare the *ClientKeyExchange* message. Note that, in case the original DTLS protocol is considered, the *pre-shared key* is directly retrieved from a locally stored set. Instead, in the presence of our security services, the *pre-shared key* coincides with the key $K_{S-C}$ previously obtained from the *TA* (see Section 6).

**Handshake duration**. Time spent by $C$ to complete the DTLS handshake.

**Transaction length**. Time spent from when the application on $C$ produces the CoAP request message to when the associated CoAP response message is received back from server $S$. Note that this encompasses also the time spent to perform the DTLS handshake. In the presence of our security services, it comprises also the initialization of additional data structures required to perform the handshake.

| | Metric | Original DTLS | Alternative DTLS |
|---|---|---|---|
| **1** | ClientHello preparation | 8.634 ms ± 0.070 ms | 9.509 ms ± 0.022 ms |
| **2** | ClientKeyExchange preparation | 4.404 ms ± 0.634 ms | 4.062 ms ± 0.446 ms |
| **3** | Handshake duration | 224.108 ms ± 2.338 ms | 206.215 ms ± 1.949 ms |
| **4** | Transaction length | 245.3 ms ± 2.396 ms | 230.95 ms ± 2.441 ms |

**Table 1** Client performance

Our results are reported in Table 1. The columns *"Original DTLS"* and *"Alternative DTLS"* refer to the original DTLS protocol and our extended implementation, respectively.

In the presence of our security services, the preliminary interaction between $C$ and the *TA* resulted in the following overhead. As to the establishment of key $K_{C-TA}$ (see Appendix B), the client experienced a round trip time equal to 22.95 ms ± 0.652 ms. Also, the client experiences a computing overhead equal to 101.373 ms ± 0.229 ms (29.436 ms ± 0.375 ms), to process the message sent to (received from) the *TA*. We recall that this key establishment is performed only the very first time that the client contacts the *TA*.

As to the actual key material exchange through messages $M1$ and $M2$ (see Section 6), the client experienced a round trip time equal to 19.35 ms ± 0.612 ms. Also, the client experienced a computing overhead equal to 0.029 ms ± 0.0002 ms and 33.465 ms ± 0.323 ms, to process message $M1$ and $M2$, respectively.

### 9.2 Server results

In this section, we present our experimental results referred to server $S$. In particular, the following metrics have been considered on the server side.

**ClientHello processing**. Time spent by $S$ to process the first *ClientHello* message received from client $C$. In case the original DTLS protocol is considered, this time actually refers to the first *ClientHello* message reception, upon which only few lightweight operations are performed. Conversely, in the presence of our security

services, this time refers to the only received *ClientHello* message, and includes the performance of anti-replay checks and the MAC verification.

**HelloVerifyRequest preparation**. Time spent by $S$ to prepare the *HelloVerifyRequest* message. This is relevant only if the original DTLS protocol is considered.

**Second ClientHello processing**. Time spent by $S$ to process the second *ClientHello* message received from client $C$, and conveying the DTLS *Cookie*. This is relevant only if the original DTLS protocol is considered.

**Check phase duration**. Time spent by $S$ between the reception of the first *ClientHello* message and the starting of Flight 4 preparation. Practically, this metric represents the time spent by $S$ to infer if the DTLS handshake is valid or not.

**PSK computation**. Time spent by $S$ to derive the *pre-shared key*, upon reception of the *ClientKeyExchange* message from client $C$. We recall that, in case the original DTLS protocol is considered, the *pre-shared key* is directly retrieved from a locally stored set. Instead, in the presence of our security services, the *pre-shared key* $K_{S-C}$ is computed from the received nonce $N$ conveyed in the *PSK identity* field of the *ClientKeyExchange* message (see Section 6.1).

**ClientKeyExchange processing**. Time spent by $S$ to process the DTLS *ClientKeyExchange* message received from client $C$. This encompasses also the retrieval or computation of the *pre-shared key*, in case the original DTLS protocol or our security services are considered, respectively.

**Handshake duration**. Time spent by $S$ to complete the DTLS handshake. In the presence of our security services, this encompasses also the update of the anti-replay sliding window (see Section 8), and the management of information for DTLS session resumption.

Our results are reported in Table 2. The columns *"Original DTLS"* and *"Alternative DTLS"* refer to the original DTLS protocol and our extended implementation, respectively. Furthermore, we provide a graphical overview of the server $S$ performance in Figure 5. The bars *"DTLS Flight 1-3 processing duration"* indicate the time spent during the *check phase* to process the first three DTLS flights.

| | Metric | Original DTLS | Alternative DTLS |
|---|---|---|---|
| **1** | ClientHello processing | 0.005 ms ± 0.0006 ms | 1.997 ms ± 0.402 ms |
| **2** | HelloVerifyRequest preparation | 0.470 ms ± 0.060 ms | – |
| **3** | Second ClientHello processing | 0.269 ms ± 0.056 ms | – |
| **4** | Check phase duration | 20.935 ms ± 1.463 ms | 10.116 ms ± 0.836 ms |
| **5** | PSK computation | 0.007 ms ± 0.0001 ms | 0.340 ms ± 0.095 ms |
| **6** | ClientKeyExchange processing | 2.084 ms ± 0.429 ms | 2.200 ms ± 0.565 ms |
| **7** | Handshake duration | 138.576 ms ± 2.220 ms | 127.359 ms ± 1.895 ms |

**Table 2** Server performance

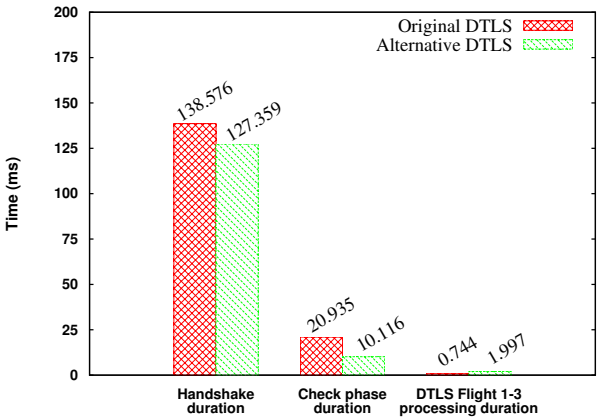


**Fig. 5** Overview of server performance

## 9.3 Discussion

In the following, we discuss experimental results reported in Sections 9.1 and 9.2. We denote as $C_i^O$ and $C_i^A$ the metrics associated to client $C$ and reported in the $i$-th row of Table 1, in case the original DTLS protocol or our alternative implementation is considered, respectively. Similarly, we denote as $S_i^O$ and $S_i^A$ the metrics associated to server $S$ and reported in the $i$-th row of Table 2, in case the original DTLS protocol or our alternative implementation is considered, respectively.

First of all, results presented in Tables 1 and 2 show that the overall handshake duration in the presence of our security services is even smaller than the handshake duration displayed by the original DTLS protocol, i.e. $C_3^A < C_3^O$ and $S_7^A < S_7^O$. The same can be observed with reference to the transaction length on the client side, i.e. $C_4^A < C_4^O$. Such results can be explained as follows. On one hand, our security services result in an extended processing of the only *ClientHello* message, both on the client and server side. Also,

as highlighted by the rightmost bar pair in Figure 5, the time required on the server side to process the only (enhanced) *ClientHello* message is longer than the overall time required to process the original two *ClientHello* messages and the *HelloVerifyRequest* message, i.e. $S_1^A > S_1^O + S_2^O + S_3^O$. On the other hand, our security services make it possible to avoid the transmission/reception and processing of two handshake messages, i.e. the *HelloVerifyRequest* and the second *ClientHello* message conveying the *Cookie*, thus reducing the overall handshake duration. Note that our evaluation has been performed in a local network, so minimazing the impact due to the communication overhead. This suggests that in a more general communication context, e.g. the Internet, our approach should result in a more relevant reduction of the DTLS handshake duration.

Second, let us refer to $T_{CA} = 52.84$ ms as the time spent by $C$ to complete the preliminary interaction with the *TA*, in the presence of our security services. This takes into account the round trip time experienced by $C$, as well as its computing overhead to process messages $M1$ and $M2$ (see Section 9.1). Then, the overall time required by $C$ to i) interact with the *TA*; ii) perform the DTLS handshake with $S$; and iii) perform the actual CoAP transaction, is equal to the sum of $T_{CA}$ and the transaction length $C_4^A$. Such a total time, i.e. 283.79 ms, is comparable with the transaction length in the presence of the original DTLS handshake, i.e. $C_4^O = 245.3$ ms. This suggests that, even considering also the preliminary interaction between $C$ and the *TA*, our security services do not substantially affect network performance from the client standpoint. We recall that such an interaction with the *TA* is required only upon starting a new DTLS session with $S$, i.e. only before sending the *first* application message.

Finally, if we focus on the server side, we observe that our security services result in a check phase which is considerably shorter than in the presence of the original DTLS handshake, i.e. $S_4^A < S_4^O$. As highlighted in Figure 5, this means that our approach asserts the genuineness of a DTLS session establishment earlier than original DTLS. Also, in the presence of the original DTLS handshake, after such a check phase, server $S$ might still be victim of a DoS attack performed with a valid spoofed IP address, which can also result in an amplification attack against other network nodes, as discussed in Section 4.1. Instead, our security services assure that, once the check phase has been completed, the current DTLS handshake is either invalid (and likely an actual DoS attack), or authentic and involving a legitimate DTLS client. As discussed above for the overall handshake duration, we believe that in a more general communication context, such as the Internet, the impact due to the communication overhead would further increase the gap between the two check phase durations $S_4^O$ and $S_4^A$, in case either the original DTLS handshake or our additions are considered, so making our approach even more advantageous.

### 9.4 Performance and effectiveness under attack

In this section, we consider an adversary who repeatedly performs the DoS attack described in Section 4.1, and present experimental results referred to when either our approach or the original one based on the Cookie exchange is adopted. In particular, we first discuss the impact of a single attack occurrence on server performance. Then, we show that our approach is consistently more effective in preserving robustness and service availability on the server side, in case a continuous DoS attack is performed.

In order to evaluate the impact of a single attack occurrence, we considered the following two metrics.

**Attack processing**. Overall time spent by server $S$ to process handshake messages during one attack occurrence. In case the original DTLS protocol is considered, this time encompasses the processing of the two *ClientHello* messages, the *HelloVerifyRequest* message, and the DTLS Flight 4. Conversely, in the presence of our security services, this time coincides with the processing of the only invalid *ClientHello* message.

**Attack exposure**. Time spent under attack by server $S$, upon a single occurrence of the DoS attack. In case the original DTLS protocol is considered, this time is measured from the reception of the first *ClientHello* message to the end of the transmission of Flight 4. Conversely, in the presence of our security services, this time is measured from the reception of the only invalid *ClientHello* message to the end of its processing.

| | Metric | Original DTLS | Alternative DTLS |
|---|---|---|---|
| **1** | Attack processing | 1.626 ms ± 0.265 ms | 1.532 ms ± 0.124 ms |
| **2** | Attack exposure | 27.860 ms ± 1.867 ms | 10.426 ms ± 1.806 ms |

**Table 3** Server performance under attack

Our results are reported in Table 3. The columns *"Original DTLS"* and *"Alternative DTLS"* refer to the original DTLS protocol and our extended implementation, respectively. We denote as $D_i^O$ and $D_i^A$ the metrics

associated to server $S$ and reported in the $i$-th row of Table 3, in case the original DTLS protocol or our alternative implementation is considered, respectively.

First of all, our approach results in overall less processing on the server side, i.e. $D_1^A < D_1^O$. Also, the processing time $D_1^A$ is shorter than the processing time $S_1^A$ in Table 2 and referred to an attack-free scenario. This is consistent with the fact that an invalid *ClientHello* message results in a reduced set of operations, i.e. the management of the anti-replay window and the DTLS session initialization operations are not performed.

In addition, the original DTLS server remains exposed to an attack occurrence for $D_2^O = 27.860$ ms, i.e. until the DTLS Flight 4 has been sent. We recall that, after that, the server maintains a half open DTLS session, and is not able to state whether it is valid or not. Instead, our alternative approach results in the server $S$ exposed for less than half that time, i.e. $D_2^A < D_2^O$. Also, after such an amount of time, $S$ is certain that an invalid *ClientHello* message has been received, and no DTLS session is maintained. As discussed in Section 9.3, we believe that in a more general communication context, such as the Internet, the impact due to the communication overhead would further increase the gap between the two attack exposure times $D_2^O$ and $D_2^A$, so making our approach even more advantageous.

In the following, we report results obtained in the presence of a continuous DoS attack performed against server $S$. Derived confidence intervals are very small and cannot be appreciated in the presented graphs. In particular, our experiments refer to the following setup.

Server $S$ was configured to maintain at most a given amount $M$ of open DTLS sessions. That is, in case such a limit is reached, $S$ does not perform any further DTLS handshake, until at least one DTLS session has been closed. We considered 300, 400, and 500 as possible values of $M$.

Furthermore, every 500 ms, a legitimate client $C$ interacts with $S$ as follows. First, $C$ performs a full DTLS handshake with $S$, so establishing a secure session. Then, $C$ performs a CoAP message exchange, after which the DTLS session is terminated with mutual agreement. Finally, the attacker host runs 3 parallel adversary processes, each one of which repeatedly performs a DoS attack against $S$ every $I$ milliseconds. We denote $I$ also as *attack interval*, and considered 50 ms, 100 ms, and 150 ms as possible values.

For the sake of proving our point with our proof of concept test environment, we considered a handshake timeout on the DTLS server equal to 30 seconds. Given the adversary considered in our experiments, this timeout effectively forces her to perform the attack indefinitely over time, in order to keep the server unavailable. This allows us to show that, in the presence of the original DTLS protocol, the considered adversary performs a successful attack which results in the server always starting invalid sessions faster than how it aborts them.

In general, the timeout should be set at least to a value $T_h$ such that the adversary is not able to start the establishment of $M$ half open sessions within a time interval equal to $T_h$. While, on one hand, smaller values of $T_h$ would force the adversary to perform a more aggressive attack, on the other hand they would make it harder to address session establishments in the presence of several, legitimate, slower clients. Note that this is not an issue when our security services are adopted, as a fake session establishment is immediately aborted upon receiving an invalid *ClientHello* message.

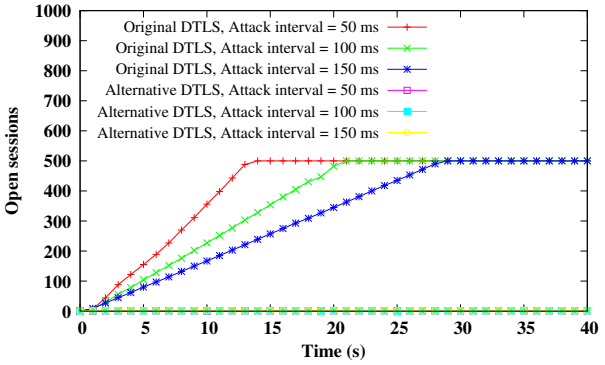


**Fig. 6** Open sessions (Max 500 sessions)

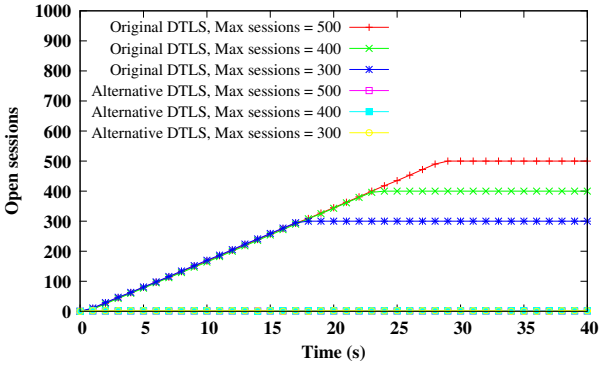


**Fig. 7** Open sessions (Attack interval 150 ms)

Figures 6 and 7 depict the number of DTLS sessions open on server $S$ over time, considering both the original DTLS protocol and our extended implementation. In particular, Figure 6 focuses on $M = 500$ maximum open sessions, while Figure 7 considers an attack interval $I = 150$ ms.

More in detail, Figure 6 shows that, in the presence of the original DTLS protocol, server $S$ keeps on opening *invalid* sessions until the maximum amount $M$ has been reached. When this happens, $S$ stops accepting new *ClientHello* messages, even if from legitimate clients. Since the adversary continuously performs the attack, new half open sessions will continuously replace the ones terminated upon handshake timeout expiration. Thus, from then on, $S$ becomes practically unavailable. Note that the time required to have $M$ (half) open sessions on $S$ is shorter when a shorter attack interval $I$ is considered, i.e. in case of a more intensive DoS attack.

Conversely, our approach promptly detects invalid *ClientHello* messages, so allowing $S$ to establish only valid sessions with legitimate clients. As a consequence, the number of open sessions remains low over time even during the DoS attack, for every considered value of the attack interval $I$.

Similar considerations hold for Figure 7. In the presence of the original DTLS, the time required to have $M$ (half) open sessions on $S$ is shorter when a smaller value of $M$ is considered, i.e. in case $S$ features a reduced amount of resources. On the contrary, our approach allows $S$ to perform only genuine DTLS handshakes, thus keeping the number of open sessions low over time, for every considered value of $M$. These results prove that our alternative approach considerably increases the robustness of $S$ in case of continuous DoS attack.

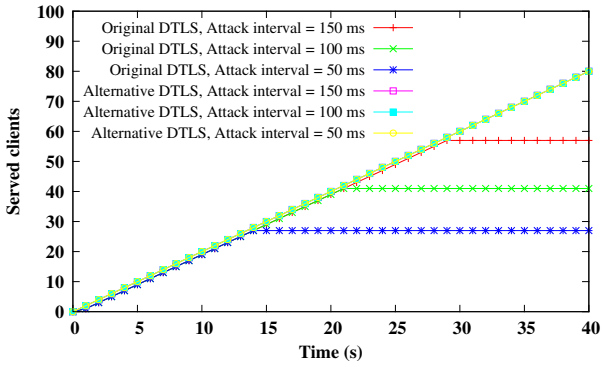


**Fig. 8** Served client requests (Max 500 sessions)

Figures 8 and 9 show the total amount of legitimate client requests served by server $S$ after a given time, considering both the original DTLS protocol and our extended implementation. In particular, Figure 8 focuses on $M = 500$ maximum open sessions, while Figure 9 considers an attack interval $I = 150$ ms.

More in detail, Figure 8 shows that, in the presence of the original DTLS protocol, the number of times that client $C$ has successfully performed a secure transaction grows until $M$ sessions have been opened on the server side. When this happens, $S$ becomes unavailable to serve even client $C$, and thus the number of served legitimate requests stops growing. Moreover, if a lower attack interval $I$ is considered, i.e. the DoS attack is more intensive, $S$ becomes unavailable after a shorter amount of time, hence further reducing the number of served legitimate requests. Conversely, our approach prevents $S$ from creating invalid half open sessions altogether. Therefore, requests from client $C$ are normally served, i.e. their number regularly keeps growing over time.

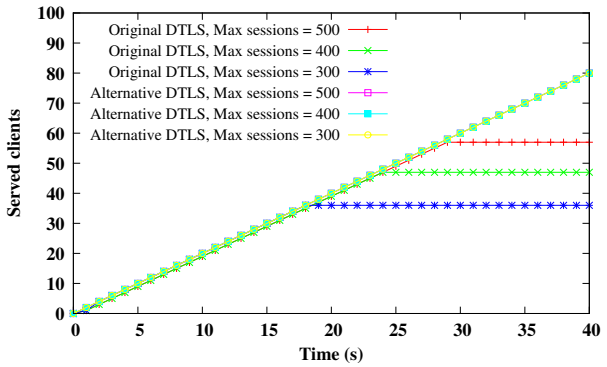


**Fig. 9** Served client requests (Attack interval 150 ms)

Similar considerations hold for Figure 9. In the presence of the original DTLS protocol, a lower value of $M$ implies that $S$ becomes unavailable after a shorter amount of time, and the number of served requests from client $C$ is consistently reduced. On the contrary, in the presence of our approach, $S$ is always able to serve requests from client $C$, i.e. their number regularly grows over time for every considered value of $M$. Such results prove that our approach effectively preserves the availability of $S$ during a continuous DoS attack.

## 10 Conclusion

In this paper, we have addressed two significant issues affecting the DTLS handshake. First, a DTLS server is vulnerable to a DoS attack aimed at starting a considerable number of half open sessions. This can exhaust memory and network resources on the server, so making it less responsive or even unavailable to legitimate clients. Second, the DTLS key provisioning based on pre-shared keys may require the server to store a considerable number of cryptographic keys. Then, it may result in scalability issues, and complicates key re-provisioning in dynamic scenarios.

We have proposed a single and efficient security architecture, which practically neutralizes the DoS attack by substantially limiting its impact, and requires the server to store only one symmetric key in case the

pre-shared key mode is used. Our approach does not require changes to the DTLS standard, does not require additional communication between DTLS client and server, and is deployable also in the TLS protocol without changing the actual standard.

Furthermore, we have presented experimental results obtained with a proof of concept implementation, and compared our approach with the original one based on the Cookie exchange. We have shown that, in the presence of a DoS attack, our approach considerably improves a DTLS server in terms of robustness and service availability. Also, it consistently reduces the time a DTLS server is exposed to an attack instance, and requires a shorter amount of time to complete a handshake execution. As a future work, we will investigate possible reactive strategies to further reduce the impact of DoS attack, upon its detection on the server side.

## Acknowledgment

The authors sincerely thank the anonymous referees and the associate editor for their insightful comments and suggestions that helped to considerably improve the technical quality of the paper. This work has been partially supported by the EU FP7 Project SEGRID (Grant Agreement no. FP7-607109) as well as by the EIT DIGITAL High Impact Initiative "Advanced connectivity platform for vertical segments".

## Compliance with Ethical Standards

This work has been carried out during the tenure of an ERCIM "Alain Bensoussan" Fellowship Programme. The research leading to these results has received funding from the *European Union* Seventh Framework Programme (*FP7/2007-2013*) under Grant Agreement no. 246016. This work was also supported by the EU FP7 Project SEGRID (Grant Agreement no. FP7-607109) and the EIT DIGITAL High Impact Initiative "Advanced Connectivity Platform for vertical segments". Ericsson holds a patent related to the *Derived Key Mode* scheme described in Section 6.1 (International Application Number PCT/SE2013/050846).

## References

1. Dierks T. and Rescorla E., *RFC 5246, The Transport Layer Security (TLS) Protocol Version 1.2*. Internet Engineering Task Force (2008)
2. DARPA, *RFC 793, Transmission Control Protocol (TCP) DARPA Internet Program Protocol Specification*. Internet Engineering Task Force (1981)
3. Rescorla E. and Modadugu N., *RFC 6347, Datagram Transport Layer Security Version 1.2*. Internet Engineering Task Force (2012)
4. Postel J., *RFC 768, User Datagram Protocol*. Internet Engineering Task Force (1980)
5. Eronen P. and Tschofenig H., *RFC 4279, Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)*. Internet Engineering Task Force (2005)
6. Scandium (Sc) Security for Californium, version 0.1.4 (2014). URL `{https://github.com/mkovatsc/Scandium}`. Last accessed: January 2016
7. Californium (Cf) CoAP framework - Java CoAP Implementation, version 0.18-2 (2014). URL `{http://people.inf.ethz.ch/mkovatsc/californium.php}`. Last accessed: January 2016
8. Shelby Z., Hartke K. and Bormann C., *RFC 7252, Constrained Application Protocol (CoAP)*. Internet Engineering Task Force (2014)
9. Constrained RESTful Environments (CoRE). URL `{https://datatracker.ietf.org/wg/core/}`. Last accessed: January 2016
10. Mirkovic J., Dietrich S., Dittrich D. and Reiher P., *Internet Denial of Service: Attack and Defense Mechanisms* (Prentice Hall/Pearson Education, 2004)
11. Wang H., Jin C. and Shin K.G., Defense Against Spoofed IP Traffic Using Hop-Count Filtering, IEEE/ACM Transactions on Networking **15**(1), pp. 40–53 (2007)
12. Li J., Sung M., Xu J. and Li L., Large-scale IP traceback in high-speed Internet: practical techniques and theoretical foundation, in *Proceedings of the 2004 IEEE Symposium on Security and Privacy* (IEEE Computer Society, 2004), pp. 115–129 (2004)
13. Savage S., Wetherall D., Karlin A.R. and Anderson T.E., Practical Network Support for IP Traceback, ACM SIGCOMM Computer Communication Review **30**(4), pp. 295–306 (2000)
14. Snoeren A.C., Partridge C., Sanchez L.A., Jones C.E., Tchakountio F., Kent S.T. and Strayer W.T., Hash-based IP Traceback, in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications* (ACM, 2001), SIGCOMM '01, pp. 3–14 (2001)
15. Song D.X. and Perrig A., Advanced and authenticated marking schemes for IP traceback, in *Proceedings of the IEEE Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2001)*, vol. 2 (IEEE Computer Society, 2001), vol. 2, pp. 878–886 (2001)
16. Ferguson P. and Senie D., *RFC 2267, Network Ingress Filtering: Defeating Denial of Service Attacks which employ IP Source Address Spoofing*. Internet Engineering Task Force (1998)
17. Li J., Mirkovic J., Wang M., Reiher P. and Zhang L., SAVE: Source Address Validity Enforcement Protocol, in *Proceedings of the IEEE Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)* (IEEE Computer Society, 2002), pp. 1557–1566 (2002)
18. Mahajan R., Bellovin S.M., Floyd S., Ioannidis J., Paxson V. and Shenker S., Controlling High Bandwidth Aggregates in the Network, ACM SIGCOMM Computer Communication Review **32**(3), pp. 62–73 (2002)
19. Yau D.K.Y., Lui J., Liang F. and Yam Y., Defending Against Distributed Denial-of-service Attacks with Max-Min Fair Server-centric Router Throttles, IEEE/ACM Transactions on Networking **13**(1), pp. 29–42 (2005)

20. Patil R.Y. and Ragha L., A rate limiting mechanism for defending against flooding based distributed denial of service attack, in *2011 World Congress on Information and Communication Technologies (WICT)* (2011), pp. 182–186 (2011)
21. Beitollahi H. and Deconinck G., A Cooperative Mechanism to Defense against Distributed Denial of Service Attacks, in *Proceedings of the IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom 2011)* (2011), pp. 11–20 (2011)
22. Dean D., Franklin M. and Stubblefield A., An Algebraic Approach to IP Traceback, ACM Transactions on Information and System Security **5**(2), pp. 119–137 (2002)
23. Goodrich M.T., Efficient Packet Marking for Large-scale IP Traceback, in *Proceedings of the 9th ACM Conference on Computer and Communications Security* (ACM, 2002), CCS '02, pp. 117–126 (2002)
24. Bhatti N. and Friedrich R., Web server support for tiered services, IEEE Network **13**(5), pp. 64–71 (1999)
25. Qie X., Pang R. and Peterson L., Defensive Programming: Using an Annotation Toolkit to Build DoS-resistant Software, ACM SIGOPS Operating Systems Review - OSDI '02 **36**(SI), pp. 45–60 (2002)
26. Juels A. and Brainard J., Client Puzzles: A Cryptographic Defense Against Connection Depletion Attacks, in *Proceedings of NDSS Symposium 1999* (1999), pp. 1–15 (1999)
27. Wang X. and Reiter M.K., Defending Against Denial-of-Service Attacks with Puzzle Auctions, in *Proceedings of the 2003 IEEE Symposium on Security and Privacy* (IEEE Computer Society, 2003), SP '03, pp. 78–92 (2003)
28. Bernstein D.J. SYN cookies (1996). URL `{http://cr.yp.to/syncookies.html}`. Last accessed: January 2016
29. Ranjan S., Swaminathan R., Uysal M. and Knightly E., DDoS-Resilient Scheduling to Counter Application Layer Attacks Under Imperfect Detection, in *Proceedings of the 25th IEEE International Conference on Computer Communications (INFOCOM 2006)* (IEEE Computer Society, 2006), pp. 1–13 (2006)
30. CERT Advisory CA-1996-21 TCP SYN Flooding and IP Spoofing Attacks (1996)
31. Mirkovic J. and Reiher P., D-WARD: A Source-End Defense Against Flooding Denial-of-Service Attacks, IEEE Transactions on Dependable and Secure Computing **2**(3), pp. 216–232 (2005)
32. Gil T.M. and Poletto M., MULTOPS: A Data-structure for Bandwidth Attack Detection, in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10* (USENIX Association, 2001), SSYM '01, pp. 23–38 (2001)
33. Wang H., Zhang D. and Shin K.G., Detecting SYN flooding attacks, in *Proceedings of the IEEE Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM 2002)*, vol. 3 (2002), vol. 3, pp. 1530–1539 (2002)
34. Ohsita Y., Ata S. and Murata M., Detecting Distributed Denial-of-Service Attacks by analyzing TCP SYN packets statistically, in *IEEE 2004 Global Telecommunications Conference (GLOBECOM '04)*, vol. 4 (2004), vol. 4, pp. 2043–2049 (2004)
35. Darmohray T. and Oliver R., Hot Spares for DoS attacks, The Magazine of USENIX and SAGE **25**(4), pp. 3 (2000)
36. Lemon J., Resisting SYN Flood DoS Attacks with a SYN Cache, in *Proceedings of the BSD Conference 2002 on BSD Conference* (USENIX Association, 2002), BSDC '02, pp. 1–9 (2002)
37. Zuquete A., Improving the Functionality of SYN Cookies, in *Proceedings of the IFIP TC6/TC11 Sixth Joint Working Conference on Communications and Multimedia Security: Advanced Communications and Multimedia Security* (Kluwer, B.V., 2002), pp. 57–77 (2002)
38. Aura T., Nikander P. and Leiwo J., DOS-Resistant Authentication with Client Puzzles, in *Revised Papers from the 8th International Workshop on Security Protocols* (Springer-Verlag, 2001), pp. 170–177 (2001)
39. Dean D. and Stubblefield A., Using Client Puzzles to Protect TLS, in *Proceedings of the 10th Conference on USENIX Security Symposium - Volume 10* (USENIX Association, 2001), SSYM'01, pp. 1–8 (2001)
40. Schneier B., *Applied Cryptography: Protocols, Algorithms, and Source Code in C, 2nd ed.* (John Wiley & Sons, Inc., 1996)
41. Cooper D., Santesson S., Farrell S., Boeyen S., Housley R. and Polk W., *RFC 5280, Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile.* Internet Engineering Task Force (2008)
42. Wouters P., Tschofenig H., Gilmore J., Weiler S. and Kivinen T., *RFC 7250, Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS).* Internet Engineering Task Force (2014)
43. Neuman C., Yu T., Hartman S. and Raeburn K., *RFC 4120, The Kerberos Network Authentication Service (V5).* Internet Engineering Task Force (2005)
44. Arkko J., Carrara E., Lindholm F., Naslund M. and Norrman K., *RFC 3830, MIKEY: Multimedia Internet KEYing.* Internet Engineering Task Force (2004)
45. Mattsson J. and Tian T., *RFC 6043, MIKEY-TICKET: Ticket-Based Modes of Key Distribution in Multimedia Internet KEYing (MIKEY).* Internet Engineering Task Force (2011)
46. Selander G. and Seitz L., *Access Control Framework for Constrained Environments, draft-selander-core-access-control-02 (Work in progress).* Internet Engineering Task Force (2014)
47. Seitz L., Selander G. and Gehrmann C., Authorization framework for the Internet-of-Things, in *D-SPAN workshop of the IEEE 14th International Symposium and Workshops on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)* (IEEE Computer Society, 2013), pp. 1–6 (2013)
48. Cole E., *Network Security Bible, 2nd Edition* (Wiley, 2009)
49. Birman K., *Guide to Reliable Distributed Systems. Building High-Assurance Applications and Cloud-Hosted Services* (Springer, 2012)
50. Anderson R. J., *Security Engineering: A Guide to Building Dependable Distributed Systems, 2nd Edition* (Wiley, 2008)
51. Shelby Z., Koster M., Bormann C. and van der Stok P., *CoRE Resource Directory draft-ietf-core-resource-directory-05 (Work in progress).* Internet Engineering Task Force (2015)
52. Krawczyk H., Bellare M. and Canetti R., *RFC 2104, HMAC: Keyed-Hashing for Message Authentication.* Internet Engineering Task Force (1997)
53. Stajano F. and Anderson R.J., The Resurrecting Duckling: Security Issues for Ad-hoc Wireless Networks, in *Proceedings of the 7th International Workshop on Security Protocols* (Springer-Verlag, 1999), pp. 172–194 (1999)
54. McElice R.J., *Finite Fields for Computer Scientists and Engineers* (Kluwer Academic Publishers, 1987)

55. Dworkin M., *Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC*. National Institute of Standards and Technology (2007)
56. Gehrmann C., Topics in Authentication Theory. Ph.D. thesis, Lund University (1997)
57. McGrew D. and Bailey D., *RFC 6655, AES-CCM Cipher Suites for Transport Layer Security (TLS)*. Internet Engineering Task Force (2012)
58. Federal Information Processing Standards Publication 180-2, *Secure Hash Standard* (2002)
59. RSA Laboratories, *PKCS #1 v2.2: RSA Cryptographic Standard*. EMC Corporation Public-Key Cryptographic Standards (PKCS) (2012)
60. National Institute of Standards and Technology, *Federal Information Processing Standards, Specification for the ADVANCED ENCRYPTION STANDARD (AES)*. National Institute of Standards and Technology (2001)

## Appendix A

In this section, we summarize the security material referred throughout the paper. Specifically, Tables 4 and 5 report the security material either pre-installed on the involved entities, or derived at runtime upon the occurrence of specific events, respectively. Also, for each entry, both tables report a brief description of the referred piece of information, as well as what network entities own it.

| Name | Description | Owners | Purpose |
|---|---|---|---|
| $K_{C-TA}$ | Long term key to interact with the *TA* | *C*, *TA* | Interaction with the *TA* |
| *seed* | Secret used for $K_{MS}$ derivation | *S*, *TA* | DoS protection |
| $K_M$ | Long term key used for $K_{MS}$ derivation | *S*, *TA* | DoS protection |
| $K_{MS}$ | Key used for $K_S$ and $K_{S-C}$ derivation | *S*, *TA* | DoS protection |
| $W$ | Anti replay window | *S* | Replay protection |
| $A$ | Anti replay window size | *S* | Replay protection |
| $w$ | Anti replay window vector ($A$ entries) | *S* | Replay protection |
| $w_b$ | Left bound of the anti replay window | *S* | Replay protection |

**Table 4** Pre-installed security material

| Name | Description | Owners | Purpose |
|---|---|---|---|
| $K_S$ | Key used for *ClientHello* authentication | *S*, *C* | DoS protection |
| $K_{MAC}$ | Key used for the actual MAC computation | *S*, *C* | DoS protection |
| $K_{S-C}$ | Derived pre-shared key | *S*, *C* | PSK provisioning |

**Table 5** Derived security material

## Appendix B

This section describes a possible approach to exchange the long term secret key $K_{C-TA}$ between client $C$ and the *TA*. The adoption of alternative procedures is, of course, possible and left open.

In particular, we denote by $N_A$ a fresh nonce randomly generated by client $C$, and by $K^+_{TA}$ the public key associated to the *TA*. Upon contacting the *TA* for the very first time, $C$ performs the following message exchange.

$$MA:\ C \rightarrow TA \quad < C, N_A, \{N_A, K_{C-TA}\}_{K^+_{TA}} >$$
$$MB:\ TA \rightarrow C \quad < \{C, TA, N_A\}_{K_{C-TA}} >$$

Since it is encrypted by means of key $K^+_{TA}$, the *TA* is supposed to be the only entity able to retrieve key $K_{C-TA}$ from message $MA$. Then, the *TA* sends message $MB$ to client $C$, encrypting it by means of key $K_{C-TA}$, and including client nonce $N_A$ retrieved from message $MA$. Upon receiving message $MB$, client $C$ verifies that the message is fresh thanks to the presence of nonce $N_A$, and has the confirmation that $K_{C-TA}$ has been correctly established with the *TA*.

In principle, $C$ can get the public key $K^+_{TA}$ from the Resource Directory service, upon contacting it to know what is the specific *TA* associated to server $S$ (see Section 5). As an alternative, $C$ can get the public key $K^+_{TA}$ from a trusted Certification Authority. Note that $C$ is required to retrieve the *TA*'s public key and establish the key $K_{C-TA}$ only once, i.e. upon contacting that specific *TA* for the first time.

## Appendix C

Although it practically takes a considerable amount of time, the *sequence number* space maintained by the *TA* and associated to a server $S$ eventually gets exhausted. When this happens, the *TA* needs to perform a rekeying with the associated server, in order to avoid reusing old session keys, which would open up for DoS attacks.

Practically, when a wrap-around of $SN$ values occurs on the *TA*, the latter temporarily stops accepting connection requests associated to $S$ from any DTLS client. Then, the *TA* generates a new random seed value $seed^+$ and a new key $K^+_{MS} = PRF(K_M, seed^+)$. From now on, the *TA* considers $K^+_{MS}$ as the new master session key, i.e. $K_{MS} \leftarrow K^+_{MS}$.

After that, the *TA* securely provides server $S$ with $seed^+$. This can be done by means of a secure channel pre-established between $S$ and the *TA*, or through an ad-hoc key distribution protocol. Further details about the actual provisioning of $seed^+$ to server $S$ are out of the scope of this paper. Finally, the *TA* resumes to regularly accept connection requests associated to $S$.

Once received $seed^+$, server $S$ computes $K^+_{MS}$ as $K^+_{MS} = PRF(K_M, seed^+)$, and sets it as the new master session key, i.e. $K_{MS} \leftarrow K^+_{MS}$. Hereafter, the *TA* and $S$ can reuse old values of $SN$, relying on the new master session key to derive session keys $K_S$.

## Appendix D

In this section, we describe how to address possible replays of *ClientHello* messages aimed at *resuming* previously established DTLS sessions, i.e. conveying a non empty *session ID* field [1]. Note that such a replay attack is considerably more difficult to be performed, as well as less convenient from the adversary standpoint. In fact, she would require to know the exact session $s$ to be resumed, especially as to the associated *session ID* value. Also, the attack would be even less effective than the one against new session establishment, because of the reduced number of handshake messages involved and their smaller size. Besides, this would also further reduce the impact on other network nodes in terms of amplification effects.

In the following, we assume that client $C$ maintains a local *resumption counter* $RC_s$ for each resumable session $s$ previously established with server $S$. Specifically, $RC_s$ is initialized to 0 after the first establishment of session $s$. Instead, on the server side, we assume that the cache table $T_S$ maintained by $S$ to manage resumable DTLS sessions (see [1]) contains also the following two pieces of information, separately for each entry $E$. First, a resumption counter $RC_s$ is used to indicate the *next* expected value of the *resumptionCounter* field in the *SecureHandshake* extension, upon receiving a *ClientHello* message aimed at resuming an old DTLS session $s$. Second, a boolean flag $F_s$ whose value is TRUE if, after the first establishment of session $s$, the key $K_{MS}$ shared with the

*TA* has not been renewed, and can thus be regularly used. Upon creating the cache entry $E_s$ associated to a just established session $s$, server $S$ initializes the $RC_s$ and $F_s$ fields to 0 and TRUE, respectively. As described in Section 8, upon receiving a new master session key $K^{+}_{MS}$, server $S$ stores the expired one as $K^{*}_{MS}$, i.e. $K^{*}_{MS} \leftarrow K_{MS}$. Also, $S$ invalidates all entries in the cache table $T_S$, by setting to FALSE their flag fields $F_s$.

Upon requesting to resume an old DTLS session $s$, client $C$ specifies the current value of its own $RC_s$ in the *resumptionCounter* field of the *SecureHandshake* extension of the *ClientHello* message. Then, client $C$ derives the key $K_{MAC} = PRF(K_S, resumptionCounter)$, which will be used for the MAC computation. After that, such a message is processed according to the same procedure described in Section 7.1, which considers also the *resumptionCounter* field as input to the *ClientHello* message authentication. Finally, $C$ sends the *ClientHello* message to server $S$.

Upon receiving such a message, $S$ retrieves the *resumptionCounter* value $RC_s$ from the *SecureHandshake* extension. In case $RC_s$ differs from the value stored in the cache entry $E_s$, the received message is assumed to be a replay and is silently discarded. Otherwise, $S$ checks the flag $F_s$ in the cache entry $E_s$, and processes the *ClientHello* message as described in Section 7.1. In particular, $S$ considers either key $K_{MS}$ or key $K^{*}_{MS}$ in case $F_s$ is set to TRUE or FALSE, respectively. As to the MAC check, $S$ relies on a key $K_{MAC}$ computed as $K_{MAC} = PRF(K_S, resumptionCounter)$. If the *ClientHello* message has found to be valid, $S$ proceeds as follows.

In case the flag $F_s$ is set to FALSE, $S$ removes the cache entry $E_s$ from table $T_S$, and replies to client $C$ with a DTLS *user_canceled* error alert message [1], so indicating that session $s$ is not valid anymore and cannot be resumed[1]. Upon the reception of such an alert message, client $C$ removes all locally stored information related to session $s$. Conversely, in case the flag $F_s$ is set to TRUE, $S$ continues to perform the DTLS handshake. Once session $s$ resumption has been completed, client $C$ increments its own local resumption counter $RC_s$. Also, server $S$ increments the resumption counter value $RC_s$ in the cache entry $E_s$ of table $T_S$.

[1] Note that the time interval between two consecutive occurrences of $K_{MS}$ re-provisioning is supposed to be much greater than the timeout considered by server $S$ for removing entries from table $T_S$.